Improving the Performance of a CAD Tool Designed to Generate Optimized Virtual Prototypes of Complete SRAM Macros

Domenic Carr, Daniel Reyno
Independent Technical Research
May 2010 – April 2011
University of Virginia
<dac7r, dar5z>@virginia.edu

ABSTRACT

This paper discusses the design, development, and testing of two additions to ViPro, a technology agnostic SRAM simulation tool. The first addition is a procedure to optimize the decoding structure for a given array topology. The second addition is a procedure to automate schematic generation of a desired SRAM configuration. We discuss design choices and implementation techniques used to make our procedures functional while at the same time attempting to minimize algorithmic complexity. We will also discuss what we learned from using the procedures. Testing was conducted using the 45nm FreePDK technology.

1. INTRODUCTION

Static Random Access Memory (SRAM) is one application of integrated circuits. Integrated circuits are very small electrical circuits, composed of mainly transistors, on a semiconductor substrate. SRAM allows users to read, write, and store strings of binary data bits, commonly referred to as words. SRAM consists of a multi-row, multi-column array that stores the data. The array is made up of bitcells, which is the actual circuit that stores the data. There is additional peripheral and control logic which assist in complete functionality of the memory. A block diagram of a typical SRAM can be seen in Figure 1.

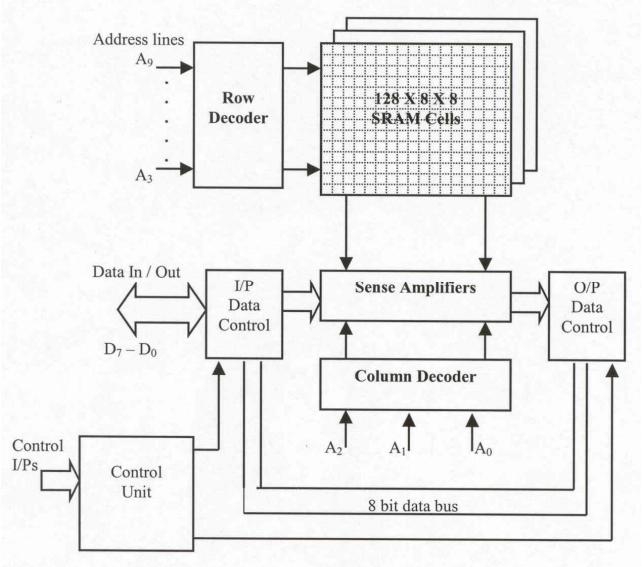

Figure 1 – Block Diagram of a Typical SRAM [1]

SRAM designs are optimized based on constraints such as delay, power, area, and yield. However, optimizing SRAM has become increasingly difficult in recent years as Moore's Law has held true, which has led to increases in device complexity due to scaling effects [2] [3]. Moreover, according to [4], design changes at any level of the design hierarchy (architectural, circuit, process) greatly affect the optimal SRAM solution since there is tremendous overhead due to scaling effects that cannot be easily estimated. Currently, however, no techniques exist that efficiently address the wide range of scaling effects at all levels of the memory hierarchy in small processes. Our project, led by Dr. Calhoun and Mr. Nalam, seeks to address this problem by designing features of the CAD tool Virtual Prototyper (ViPro) that will allow users to evaluate virtual SRAM prototypes in terms of the characteristics listed above at every step of the design process.

Currently, ViPro is in the initial design stages. Mr. Nalam has already built the initial framework of ViPro that determines how the various aspects of the tool will interact. ViPro allows the user to specify SRAM performance parameters. ViPro then iterates through circuit tests to determine an optimal design based on the user-specified constraints. The final output is a virtual prototype of the optimal design based on the design constraints. Figure 2 shows the tool flow of ViPro and how it operates to deliver an optimal design: user inputs to optimization to virtual prototype.

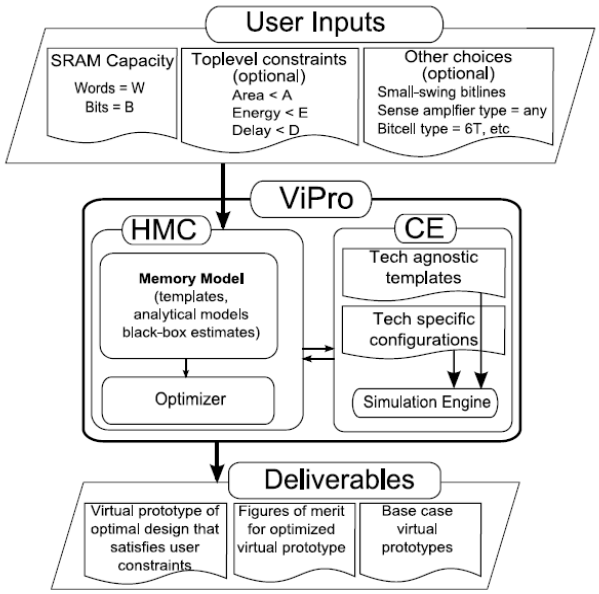

Figure 2 – ViPro Tool Flow [4]

To advance his work, we added design capabilities for two main parts: decoder optimization and automated generation of circuit schematics. To do this, we wrote code in various languages and performed extensive computer simulations on many SRAM designs. The first section of this paper is dedicated to the decoder optimization feature. The second section of the paper is dedicated to the automated generation of SRAM schematics.

2. DECODER OPTIMIZATION

2.1 Introduction

Decoder optimization is an important aspect of SRAM designs because the path through the decoder is often on the worst-case path, which limits the performance of a memory. This section discusses general techniques used to optimize a decoder and proposes a method to systematically identify an optimal design for a given array topology. The decoder optimization techniques presented here are associated with a single-bank SRAM architecture. The tradeoff between delay and power/energy were the two primary figures of merit focused on. The method to identify the optimal decoder design was implemented in a computer program, which automated the optimization procedure and will allow it to serve as a plug-in for the CAD tool ViPro.

The purpose of developing this optimization procedure was to add functionality to ViPro. Thus, the design of the program was done in a way so that it can be accessed and controlled by an upper-level program, although it can also function as a stand-alone program. This means that all of the tests that would be performed in an upper-level program can be applied to the decoder optimization procedure at the lower level. Put another way, if one is interested in sweeping across certain variables (e.g. operating temperatures, VDD values, etc.) for the entire memory, those sweeps can be done for the decoder optimization as well, resulting in an optimal decoder structure according to those constraints. The actual implementation of the algorithm for optimizing the decoder in code presented numerous challenges that led to interesting findings that will be detailed later in the section.

The decoder that was tested and optimized was a static implementation in the 45nm FreePDK technology. However, the code was written in such a way that it can be generalized to simulate different decoder schematics. That is, by following the guidelines provided in [5], the "Decoder Optimization Plug-in for ViPro User Guide", one can include his or her own netlists of a different decoder implementation (e.g. dynamic) and run the program to obtain the delay and power results. Moreover, the program was written in a technology agnostic way, such that it can be easily converted into a technology agnostic simulation environment, which will allow for its use with other process technologies, not just the 45nm FreePDK technology.

This section will begin with the background research that led to the ideas on how to choose and alter certain parameters of the decoder structure to vary the delay and energy. Then, a discussion will follow on how these parameters are affected by the constraints of the global architecture of the SRAM in ViPro. Next, a description of the decoder model generated will be discussed. Finally, the testing of the decoder to obtain the optimal structure will be presented and then an evaluation of the testing results will follow.

2.2 Background Research

Understanding the energy-delay tradeoff for decoders is a well-researched problem. [6] used a logical effort approach when analyzing the critical path through multiple-level decoders and determined that an important parameter to consider for power dissipation is the ratio of the word driver input gate capacitance to the predecoder wire capacitance. The larger this value is, the more power dissipation. Conversely, the smaller this value is, the less power dissipation. When combined with sensitivity analysis from [7], that the savings in one variable corresponding to a relaxation in another is greatest when operating toward an extreme, we can see that by relaxing the constraints of logical effort sizing for minimum delay we can greatly reduce power consumption. Similarly, by increasing the ratio of the word driver input gate capacitance to the predecoder wire capacitance to have a slightly larger power dissipation, a larger gain in speed (decrease in delay) can be expected.

These parameters manifest themselves in this application when varying the rows and columns for a given size memory. For example, in a fixed size memory, increasing the number of rows decreases the number of columns. Therefore, the increase in the number of rows increases the height of the memory, which increases the predecoder wire capacitance. Furthermore, the decrease in the number of columns decreases the number of bitcells loading the word line, thus decreasing the word line capacitance, which in turn requires a smaller word driver input gate capacitance. Therefore, we have two effects of an increase in the number of rows simultaneously driving down the ratio of the word line capacitance to the predecode wire capacitance. By similar logic, a decrease in the number of rows will have a similar dual effect when increasing the ratio. From this, it can be seen that manipulating the aspect ratio of a given size memory is a strong factor in adjusting a decoder's power consumption.

To alter the delay, the number and sizing of buffers plays an important role. Using the results of logical effort analysis, a fanout of 4 is the optimal sizing for minimal delay. Thus, including a buffering system comprised of fanout-4 inverters leading into the predecode capacitance and/or word line driver capacitance will decrease the delay of the decoder transition compared to an unbuffered path. As seen from the preceding analysis, the values of load capacitances along the decode path change as the aspect ratio of the memory are changed. Thus, the number of required buffers to optimally drive these loads will change as well. This provides yet another parameter to manipulate when searching for an optimal decoder design.

The background research, therefore, provided a strong foundation for developing an algorithm to optimize the decoder structure for a given array topology (fixed sized memory). Namely, adjusting the number of rows/columns, number of buffers before the predecode wire and number of buffers before the word line driver all provide different ways to manipulate the delay through and power dissipation of a decoder. By playing with these parameters, the intention was to generate a pareto optimal curve of decoder energy/delay values.

2.3 Overview of SRAM Architecture

With ViPro having a single-bank architecture and with its corresponding 8-to-1 output column mux, each row in the memory is constrained to having only 8 words. Therefore, the maximum number of bitcells in a given row is 8 times the word

size. The capacitance of this load is the maximum capacitance of the word line for a given column/row combination. An assumption the optimization procedure makes is that there is always a non-jagged, rectangular array of bitcells. That is, each row will have the same number of words. Furthermore, the number of words in a column can only be a power of 2 (1, 2 ,4 or 8, due to a maximum of 3 column address bits). Thus, the capacitance for a given row/column combination is always limited to the values corresponding to a capacitance of either 1, 2, 4 or 8 times the word size number of bitcells.

Due to ViPro having a single-bank architecture, the predecoder wire length is simply the height of the single array of bitcells. The height of the array of bitcells depends on the number of row address bits. For N row address bits, there are 2^N rows in the array. Therefore, the predecoder wire length is 2^N times the height of a bitcell. The height of a bitcell is a process-dependent value, and is taken into account in a parameters file that the optimization procedure reads in at the onset of the procedure. Then, based on this length, the capacitance and resistance of the predecode wires can be computed by multiplying the predecode wire length with the capacitance and resistance values per unit length (process-dependent values that are also in the same parameters file). Thus, we can generate the parasitic wire capacitance and resistance for any size memory by simply using the number of row address bits.

2.3.1 Row and Column Address Bits

ViPro is limited to 9 row address bits. Thus, the maximum number of rows in the array of bitcells is $2^9 = 512$ rows. Then, to maximize the number of words per row, all 8 word slots should be used, resulting in the maximum size of a memory of 2^9*2^3*word size. For example, if we take word size as 32, or 2^5, then the maximum size memory is $2^9*2^3*2^5 = 2^{17}$, or 131072 bits. On the minimum side, ViPro could theoretically use only 1 row address bit, but that is a highly unusual case, so another assumption the optimization procedure makes is that there will never be fewer than 3 row address bits. For the case of a 32-bit word, the minimum capacity memory would be $2^0*2^3*2^5 = 2^8 = 256$ bits. This leaves a range of possible memory sizes from 256 to 131072 for a 32-bit word in this architecture.

It can be seen that for a given size memory the number of row bits and column bits can vary. This is where adjusting the aspect ratio of the memory plays into decoder optimization. With the constraints that the row address bits, r, can never fall below 3 or exceed 9, and for a give sized memory of 2^N,we can write an equation that expresses this relation:

$$2^r*2^C = 2^N \text{ , or, } r + C = N, \text{ subject to } 3 \le r \le 9 \qquad (1)$$

Where, C = number of column bits + $\log_2$(word size)

By substituting C = c + w into (1), the equation simplifies to:

$$r + (c + w) = N, \qquad 3 \le r \le 9,\ 0 \le c \le 3 \qquad (2)$$

Then, rearranging,

$$r + c = N - w, \qquad 3 \le r \le 9,\ 0 \le c \le 3 \qquad (3)$$

Where, c = number of column bits
w = $\log_2$(word size)

Now, to continue with the case of a 32-bit word (w=5), it was shown that the possible memory sizes range between 256 and 131,072, which correspond to N values of 8 and 17 respectively. Thus, (3) simplifies to the following inequality:

$$3 \le r + c \le 12, \qquad 3 \le r \le 9,\ 0 \le c \le 3 \qquad (4)$$

The inequality in (4) represents the range space for possible aspect ratios that the decoder optimization procedure will sort through when determining power and delay combinations.

To see how r and c can be varied over a set space to change the aspect ratio and alter the delay and power through the decoder, consider the following example of 16,384-bit memory (N=14) in an architecture using a 32-bit word size (w=5).

Using (3), $r + c = 9$, subject to: $3 \le r \le 9,\ 0 \le c \le 3$.

This results in the following (r , c) combinations:

(6 , 3) ; (7 , 2) ; (8 , 1) ; (9 , 0).

Each of these r-c combinations will provide a unique combination of word line capacitances and predecode wire capacitances, which in turn will require different optimal number of buffers along the decode path. Thus, to obtain the optimal decoder structure for 16,384-bit memory, these are the r-c combinations that need to be examined.

A final note on the r-c constraint is that since c can only vary between 0 and 3, the maximum number of combinations that can ever be examined for a given size memory is 4.

2.3.2 Predecoding and Word Line Generation

ViPro is currently designed to have a set decoder structure. That is, there will always be three predecoders that take in 3 row address bits each, thereby generating 24 predecode wires (8 each), which are then combined to generate the proper word lines. Therefore, a requirement of this structure is that 9 row address bits have to be input into the decoder optimization procedure, even when a case of fewer row address bits is being analyzed. For these cases, the non-needed inputs are simply zero. Thus, certain predecode wires will never evaluate to a 1, and are simply not used when generating the word lines. For example, if all three inputs of a predecoder are 0 (occurs for the cases when r = 3 or r = 6), then the predecode output of 0 is the only predecode value that would ever evaluate to a 1. Thus, the predecode 0 output would be the only predecode wire connected to the remainder of the decoder structure. The remaining 7 predecode outputs still exist, but do not connect to anything. They will therefore not contribute to any of the decoder power or delay. The 1 “hot” predecode output will also not contribute any power, because it is simply VDD for all time, and thus will not draw any current from the supply. Having to use this predecode wire in the generation of the word lines, however, will add some delay to the decoder (as compared to the case where non-necessary inputs are not included) because it has to be ANDed with the lower order predecode wires. Since the value of the predecode wire is a

constant VDD though, the effective addition of delay to the decoder worst-case path would be equivalent to 2 min-sized inverter delays.

Each predecoder is currently designed the same way. The 3-to-8 predecoder used in the optimization procedure has 3 inverters and 12 AND gates. All gates are static. The 3 inverters generate the complements of the inputs, 4 AND gates decode the upper 2 inputs, and the remaining 8 AND gates generate the 8 outputs. All gates were sized for equal pull-up and pull-down resistances, so that the switching threshold was at VDD/2. This sizing is something that can easily be changed in the netlist file, and is a knob that can be adjusted in future work.

As stated earlier, only potentially active predecode wires are connected in the decoder structure to generate the word lines. For the highest order combinations (the combinations of outputs from the two most significant predecoders), there will always be 8 redundant lines generated. That is, each unique combination of potentially active upper and middle predecode wires will have 8 identical outputs sent into AND gates to combine with the lower predecode wire. This means that each upper-middle AND output will only drive 1 AND gate. Figure 3 shows a diagram of the predecoding and word line generation in the ViPro decoder.

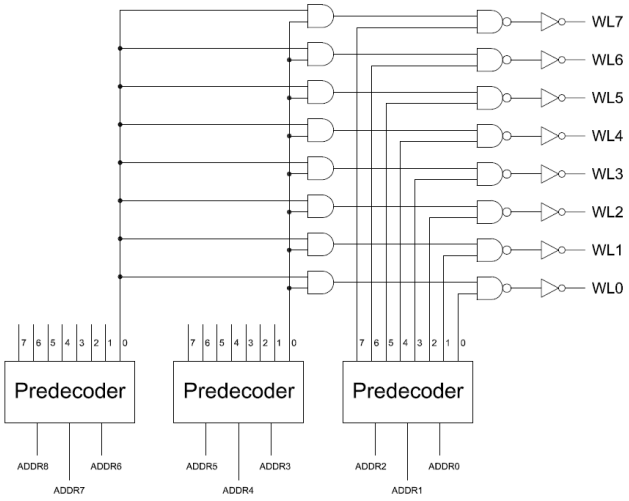

Figure 3 – Predecoding and Word Line Generation

To illustrate this idea, consider the case of a 6-bit decoder. The upper 3 row address bits are all 0. Thus the upper predecoder will only ever have one active predecode wire, the 0 output. This wire is then the only upper predecode output connected to the 8 middle predecoder outputs. However, instead of only having 8 AND gates for the 8 unique upper-middle combinations, there are actually 64 AND gates - 8 AND gates for each of the 8 unique combinations (i.e. upper0 AND middle<n>, for n=0,1…7 is generated 8 times). Then, each of these wires drives one AND gate (the other input to that AND gate is from the lower predecoder) and the 64 unique word lines are generated.

2.4 Description of Model

An essential part of ViPro will be to quickly analyze the effect of design changes on important figures of merit of the SRAM. The decoder optimization procedure reports two figures of merit: power consumption and delay. In order to report these values in a timely manner, it was essential to develop the smallest model possible that could be manipulated to accurately represent the entire decoder. Otherwise, simulating an entire decoder with its thousands of internal nodes would take too long and would not be useful for ViPro.

The model developed to simulate the delay through the decoder is simply the worst-case path through the decoder. Thus, the model consists of the gates that lie on the worst-case path loaded with their proper input and output capacitances. The worst-case transition for the decoder is switching from all 1's to all 0's. So, to measure the delay, we simply measure the time it takes for word line 0 to rise to VDD/2 after the input falls to VDD/2, or its t_{plh} value.

To measure the power consumption of the decoder, it was important to try to re-use the worst-case path, since we were already using that as the model for the delay. Having to create an entirely new netlist and run a completely new simulation would be very redundant and does not fit in the future vision of ViPro delivering quick analysis. So, the worst-case path was used as the model for the power consumption as well by exploiting the methods explained in [8]. In essence, power consumption through a decoder is described by a geometrically-decreasing amount of active, or switching, paths through decoder. The worst-case path is one of those active paths. Thus, to obtain the full decoder power from the worst-case path, certain gates along the path should be multiplied by the amount of paths that are active in that stage. Put another way, each gate on the worst-case path should be multiplied by the number of other gates in that same stage of the full decoder that switch from ground to VDD.

To illustrate this with a simple example, consider a simple 4 to 16 decoder that transitions from 1111 to 0000. The worst-case path would include one inverter, followed by two ANDs (one predecode and one output). Since there are 4 inverters in the full decoder that experience a 0 to 1 transition, the first inverter should be multiplied by 4. The predecode AND gate should be multiplied by 2, since there are 2 predecode AND gates that transition from a 0 to a 1 (these are the AND gates that decode 00). Finally, the output AND gate is simply multiplied by 1, since there is only one active output (the 0000 output).

By extending the method explained above to ViPro's architecture, the worst-case path can be used as a model for the power consumption of the entire decoder. Thus, the worst-case path serves as both the means of obtaining the delay and power for the decoder. This, then, is the basis for how the decoder optimization procedure can determine power and delay results for a given decoder structure.

2.5 The Decoder Optimization Procedure

The decoder optimization procedure is a program that combines code from various computer languages. At the top-most level, a PERL script determines the desired type of simulation, calls Ocean in the appropriate directories, and runs the simulation. The middle level is the Ocean file, which contains the code that controls the Cadence software. The bottom level is the netlists that contain the subcircuits being tested (based off the model discussed in section 2.4). The netlists are located in subdirectories that are specific to the number of row address bits. Thus, there are 7 subdirectories (named 3bits, 4bits, 5bits … 9bits).

Essentially, the procedure allows a user to sweep across all possible row/column combinations for a fixed-sized memory, sweep across the number of buffers (n) before the predecode wires and the number of buffers (k) before the word line for each row/column combination. The procedure then reports the power and delay results for all of the sweeps in a text file. It is then up to ViPro, to read in this text file and determine which power-delay values match up best with the metric it is trying to minimize. How ViPro chooses the appropriate combination is out of the scope of this project. This procedure simply produces the menu of possible power-delay choices, which then allows the decoder to be optimized for whatever metric is desired.

The PERL script does not take in command-line arguments. It is simply an executable whose process flow is determined by flags at the start of the file. These flags must be changed prior to runtime in order to run the desired type of simulation. There are three types of simulations: (a) Sweep across all possible row/column combinations and all n/k buffer values; (b) Sweep across all n/k buffer values for a given row/column combination; (c) evaluate a particular row/column, n/k buffer value combination. Based on the values of the flags in the PERL file, one of these three simulations will be run.

Next, once the desired simulation has been chosen, the PERL script invokes command-line arguments to enter the appropriate subdirectories to simulate the desired models. When it does this, it calls ocean to start. When ocean is started, it is told to immediately load the .ocn file, and then exit ocean once the .ocn file is completed. After this process is completed in the necessary subdirectories, the PERL script compiles all of the results from each subdirectory into one results file.

There is only one .ocn file. It resides in the main directory, and it is copied into each necessary subdirectory by the PERL script. The advantage to having only one .ocn file is that the code is confined to one place. This means that when making a change to the simulation, it is not necessary to change code in multiple directories, specific to only one row/column case. Using only one .ocn file dictated that the .ocn file be a very generic file that is heavily dependent on parameters it reads in. The parameters it reads in provide the means for making the .ocn file specific to a particular row/column case.

There are two sets of parameters that the .ocn file must read in: global and local. Global parameters are values that each row/column case share in common. Global parameters are meant to be configured by the user. Examples of global parameters are the capacity of the memory, the parasitic resistance of the predecode wires, and the parasitic capacitance of the predecode wires. Local parameters are values that are specific to the number of row address bits the decoder structure has. Local parameters are fixed values that only developers of the program should adjust (this should rarely happen). Examples of local parameters are the number of row address bits that each subcircuit in a given row address bit subdirectory has (i.e. the 9bit subdirectory would have a local parameter of r = 9), and the energy multipliers each subcircuit in a given row address bit subdirectory shares. The global parameters file is located in the main directory. Each subdirectory contains its own local parameters file.

Once these files have been read into the .ocn file, there is only one more step before simulation can begin: determining whether a sweep of n and k is desired or not. To do this, the .ocn file reads in a temporary file created by the PERL script that says whether or not the .ocn file should do a sweep or not (the temporary file contents are generated by the flags in the PERL script). Once this has been determined, the simulation begins.

The first thing the simulation checks is whether the given number of row address bits is compatible with the memory capacity. This is done by employing the method in section 2.3.1. If, for a certain capacity, the number of row address bits in a given subdirectory is incompatible (i.e. equation (3) is not satistifed), the simulation ceases and control will be given back to the PERL script. If the number of row address bits is compatible, then a 32 ns simulation is run on the associated netlist file, regardless of whether a sweep or individual test is being performed. After the simulation is complete, if a sweep was the desired simulation type, then each subcircuits's power-delay results are included in the subdirectory's output file. If an individual test was the desired simulation type, then only that combination of n and k will be put in the subdirectory's output file.

The netlist file contains the subcircuits that are used to build the models tested by the ocean file. Subcircuits are parameterized so that it is easy to change certain values in the subcircuits (e.g. widths/lengths of transistors, etc.). The most important subcircuits are those that correspond to the various combinations of n and k values. This procedure evaluates 9 combinations of n and k values (n = i, k = j; for $0 \leq i,j \leq 2$). These subcircuits are referred to as DUT<n>, for n=0…8 in the .ocn file, and they are the only subcircuits ever referred to directly by the .ocn file. Each of these subcircuits has a rigid structure comprised of the following devices: inverter, predecoder, nbuffers, predecode wire, nand, inverter, word line driver, kbuffers. (Note: there is one exception. The subcircuits in the 3-bit netlist only lack the nand and second inverter devices, but this is detailed in the user guide). The reason for the rigid structure is so that the .ocn file can call the same device name in each subcircuit across all row address bits when requesting the power consumption for each device. Otherwise, if each subcircuit had a variable list and number of devices, the .ocn file would not be able to handle acquiring the power values without hard-coding. Therefore, this method allows the program to acquire power values for each model while hiding the implementation of the devices that comprise the model.

When n=0, it means that there are no buffers before the predecode wire. When n=1, there is one buffer before the predecode wire. When n=2, there are two buffers before the predecode wire. Each buffer is comprised of an inverter driving a fanout 4 inverter. Thus for n=1, the inverters that make up the buffer are of sizes 1- and 4- times a minimum sized inverter. For n=2, the inverters are of sizes 1-, 4-, 16-, and 64- times a minimum sized inverter. The same convention holds for the k buffers.

Thus, each row address bit simulation provides 9 unique power-delay points when a sweep of the n,k values is performed. When a sweep across rows/columns is done as well, a maximum of 36 power-delay points is generated. ViPro can then use all of these points to determine which is the optimal for a given metric.

2.6 How Testing is Done

Running the decoder optimization program is very easy to do. The only requirement before running the program is that the user must have started cadence. Other than this, use the command-line to switch to the main directory (called SweepRows) which contains the SweepRows.pl file. From within this directory, the

command "perl SweepRows.pl" starts the execution of the program with the default flag values and the default global parameter values. To change the flag values, simply open the .pl file in a text editor, and change the values according to the comments in the file. To change the global parameter values, open the "params" file in a text editor, edit the values, then save the file. The typical reasons to adjust the "params" file would be to change the capacity of the memory, or to change the process-dependent values (parasitic resistances, capacitances, bitcell height, etc.). After running the perl command, the "results.txt" file that appears in the main directory will contain the results of the simulation. (Note: once the perl command is executed, if a file with the name "results.txt" exists, it will be overwritten. Rename the "results.txt" file after completing a simulation to prevent losing data).

2.7 Evaluation of Testing

After completing the decoder optimization program and testing it on various sized memories I gathered all of the results files and plotted the energy versus delay values in excel. Energy was calculated by taking the power value output from the procedure (average power) and multiplying it by the length of the simulation (32 ns). As stated earlier, we had hoped to find a Pareto optimal curve in the energy-delay space. However, upon plotting the results, I did not find a Pareto curve. Instead, I found data that looked like the plot in Figure 4:

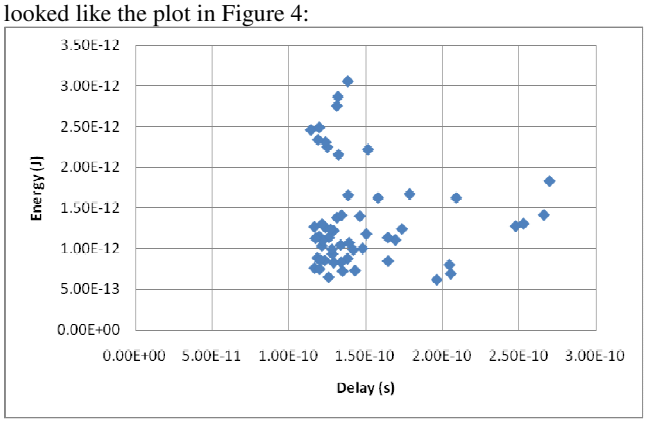

Figure 4 – Energy vs. Delay for a 16,384-bit memory

There does seem to be some type of curvature to this data, but surely not one that looks like a Pareto curve. It is possible, perhaps, that a Pareto curve should not be expected when simulating across different sized decoders (i.e. different number of row bits). However, if a Pareto curve is supposed to be attainable, a possible source of error would be from inaccurate power calculations. When verifying the model with a full decoder, I was able to match the delays in the model with the delay in the full decoder. I was able to do this for various row address bit cases. However, I was not able to accurately use the model to predict the *full* power of the decoder on a consistent basis. For smaller memories, the power of the decoder was often close to that reported by the model, but for larger memories this was not the case. Going stage-by-stage in the full decoder and comparing it to the model, I was able to verify that the energy multiplier method I use works to obtain the active power. For cases when the active power is much larger than the non-active power, the method I use to calculate power is accurate. However, for cases where this is not true, the power is not accurate. Specifically, the power would be underestimated. Thus, looking at Figure 4, if points toward the bottom left of the cluster were the underestimated power points, then by shifting these up to larger power values could result in a more Pareto-looking curve.

Figure 4 does, however, provide the data points (and hence n,k combinations) that work best for the decoder. The best data points would be those with the least power dissipation and the least delay. In Figure 4, these points correspond to those in the bottom left corner. Typically, to achieve these points, the buffer values would include an n=1 and/or k=1. For memories with larger height, n would normally be 1, and for memories using up all 8 words in the column, k would normally be 1. When these cases overlap, (for larger memories) the optimal buffering system would be n=1, k=1. Now, recall that a buffer value of 1 means the load is being driven by a 4-times minimum-sized inverter. These are quite small buffers. These are most likely the optimal buffers for the memories examined in this project because of the constraints of the ViPro architecture described previously. If the ViPro architecture were to change (i.e. allowing more rows and columns to create bigger memories) then it is very likely that higher orders of n and k would be included in the optimal cases of buffering since there would be bigger loads to drive. In this architecture, the loads are simply too small to be driven by larger buffer networks, thus the delay is worse when using the larger buffer networks.

Other than a Pareto curve not developing from the data, the work I am pleased with the work I accomplished. The program I wrote provides a very solid infrastructure for testing any type of decoder, provided the guidelines in the user guide are followed. The program offers a lot of flexibility in that it is very general, and there is very little hard-coding. Abstracting the implementation of the models allows someone to insert their own subcircuits and test a completely different decoder. However, there is an inherent rigidity to the testing in that if someone wants to test different values of n and k, other than 0, 1, and 2, new subcircuits have to be added to all 7 netlist files and the .ocn file has to be altered to include the measuring of the new subcircuits. This is because there was no way to parse the language in the .ocn file such that you could iterate over n,k values and concatenate them to a string. If this were possible, the .ocn file would not have to "manually" (hard-code) iterate through the different n,k combinations.

For future work, I would suggest adding an n,k=1.5 stage of buffers. By moving from 1 to 2 in the n,k values, the final inverter of the chain of buffers changes from 4 to 64. Thus, my analysis never gets to see the results of a sized 16 inverter driving the predecode wires and/or word line. These could provide more data points that would be of interest, since 16 seems like it would be more in the area of an optimal design for the type of load capacitances we are dealing with (4 feels too small and 64 feels too large). I have performed limited analysis of the n,k=1.5 stage of buffers and the newly-added points to the power-delay space did not add anything toward a Pareto Curve. As can be seen in Figure 5 on the next page, the newly added points (the circles) to the space are primarily in the bottom left corner.

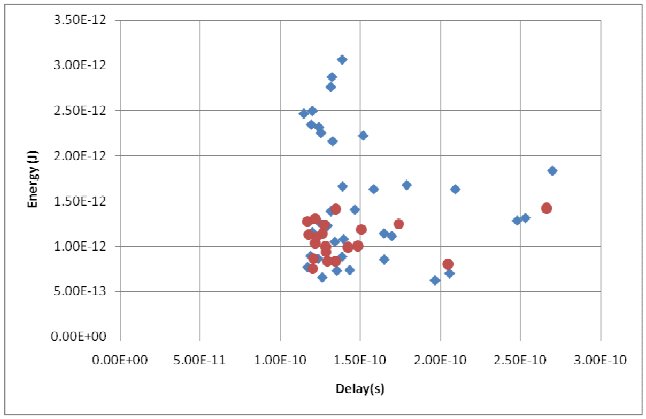

Figure 5 – Energy vs. Delay after adding n,k=1.5 values

However, even though these points are no closer to forming a Pareto curve than the previous data set, including buffers with the final inverter of size 16 would provide good data points to have in the optimization analysis, since it provides a less drastic jump in the data space. Moreover, these points also appear in the bottom left corner, which was identified earlier as the area of optimal combination of delay and energy, so this further supports their use in the optimization procedure.

I would also suggest varying the fanout of the inverters. Instead of using only fanout-4 inverters, it would be interesting to sweep a range of fanout values building up to 4. Thus, parameterizing the fanout values would be a good idea, and then including this in the sweep would provide even more values in the power-delay data space.

Another suggestion would be to sweep across the sizing used in the decoder. This should be very easy, since the sizes of all the elements used in the models have been parameterized. Thus, to do this would only require adding a variable to sweep across in the .ocn file. This is also a "finer" knob to adjust when trying to tweak the energy-delay values. Thus, using this knob could help develop the Pareto optimal curve better than only sweeping across the buffer values on the worst-case path, since adjusting the buffer values is a much more "coarse" knob. Another reason to sweep sizing is that sizing up the elements in the decoder would decrease the delay through the decoder. Even though power consumption would increase due to sizing up the elements, decoder power consumption does not constitute the major source of power consumption in a full SRAM (it probably consumes around 20% of total SRAM power). Thus, it would be of interest to generate more points that have smaller delay values so that SRAM designers can choose from points these points when designing their SRAM. These points would be appear further up and to the left (northwest) in Figures 4 and 5.

With regard to the code I wrote, I would recommend that a place for work would be in the PERL script. Making the flags adjustable at runtime would be a good idea. A way to make this happen could be to set the flags as command-line arguments. Then, by running the perl command with different command-line arguments, you could choose what simulation you want to run at runtime. This would be extremely helpful for ViPro, especially since the upper-level scripts of ViPro will most likely be calling the decoder optimization plug-in by means of the command-line. The global parameters file, "params" is currently a temporary solution for having the decoder optimization procedure have the ability to be configurable. In the future, these parameters would be contained by ViPro at a higher-level of the program hierarchy. Thus, the optimization procedure could either call directly to that file, or, ViPro could use a PERL script to copy the necessary values from the "highest-level" parameters file of ViPro and print them in the "params" file in the SweepRows directory. Both methods would work, so choosing one way over the other would be a matter of preference to the overall designer of the system.

3. SCHEMATIC GENERATION

3.1 Introduction

The automatic generation of schematics occurs in a hierarchy structure. The basic gates and blocks are first created and CDF parameterized, if necessary. Then, upper level scripts create intermediate circuit components, called leaf nodes. Finally, a top level script connects all the leaf nodes together, creating the entire SRAM schematic. All schematics are created within the Cadence simulation environment, using the SKILL scripting language. One feature of this hierarchy structure is that only a single SKILL script must be loaded in order to generate the complete SRAM schematic. Also, the scripts are technology agnostic; they can generate the entire SRAM schematic for any given technology node. An important feature of generating schematics is specifying the FET sizes for any desired component within the SRAM. These scripts allow users to have a parameters file which contains desired sizing and the procedures will automatically generate the SRAM schematic using the sizes specified within the file. If no sizing for a certain component is specified, minimum sizing of the technology node is used. The following sections will describe the hierarchy structure of the schematics of the SRAM.

3.2 General Notes

There are some general notes to discuss that pertain to all gates and components of the SRAM. All circuits are placed within a single user-specified library in the Cadence simulation environment. Only the FETs must be accessed from a different library. All circuits have input/output pins for power and ground (called VDD and VSS, respectively). Also, all procedures require that the sizes passed to the procedure must be of variable type *string*, as opposed to a *float* or some other variable type. All scripts use and build upon the existing scripts developed by previous members of the ICGroupSRAMTool research team. The "p" designation for widths/lengths refers to PMOS FETs and the "n" designation for widths/length refers to NMOS FETs, i.e. "wp" would be the width of a PMOS while "wn" would be the width of an NMOS. For additional reference concerning any portion of the schematic generation procedures and examples of how to use the specific procedures, please see [9], the "Schematic Generation User Guide."

3.3 Basic Gates, Components, and Blocks

Initially, the lowest level gates, components, and blocks are created. This includes gates that will be reused in multiple places, i.e. inverters, NANDs, etc., within the SRAM, components that will be used to create larger circuit components, i.e. the bitcell for use in the bitcell array, and blocks, such as the timing block. For the gates that will be reused, they must also be CDF parameterized, by creating CDF parameters for the devices within these gates. By creating CDF parameters for re-used gates, it eliminated redundant schematic generation. The parameterization occurs within the procedure that generates the schematics for the

given gate. The gates that must be CDF parameterized are the inverter, buffer, 2-input NAND, 2-input AND, 2-input NOR, and the tri-state inverter. The CDF parameters are the widths and lengths of all the FETs within the gate. Since these gates will be reused, they are given minimum sizing associated with the given technology node. The basic circuit components created during this process are the bitcell, D Flip Flop, buffer chains (used within the timing block), column MUX, and sense amplifier. The blocks created are the eight word line driver and the input/output block. Since the components and blocks will only be used once, or will be the same one used over and over again, they do not need to be parameterized. However, their sizing must be set during their initial creation, or else the FETs within these gates will be set to minimum size for the given technology node.

3.3.1 Inverter

This is the typical static CMOS inverter. It has the following parameters: wp, wn, lp, and ln.

3.3.2 Buffer

The buffer is two series inverters. It has the following parameters: wp1, wn1, lp1, ln1, wp2, wn2, lp2, and ln2. The "1" designation refers to the first inverter and the "2" designation refers to the second inverter.

3.3.3 NAND

This is the typical static CMOS 2-input NAND gate. It has the following parameters: wp, wn, lp, and ln. An important note is that the specified size for the NMOS devices is what both NMOS FETs are set to, i.e. if wn is set to 100n, both NMOSs have widths of 100n – the procedure does not account for two FETs in series on a path. For further reference, see Schematic Generation User Guide.

3.3.4 AND

This is the typical static CMOS 2-input AND gate. It has the following parameters: wpNAND, wnNAND. lpNAND. lnNAND, wpINV, wnINV, lpINV, and lnINV. The "NAND" designation refers to the NAND portion of the AND gate while the "INV" designation refers to the inverter portion of the AND gate. As with the regular NAND gate, the size of the NMOS in the NAND portion of the AND gate is set to whatever is specified by the user – the procedure does not account for two FETs in series on a path.

3.3.5 NOR

This is the typical static CMOS 2-input NOR gate. It has the following parameters: wp, wn, lp, and ln. An important note is that the specified size for the PMOS devices is what both PMOS FETs are set to, i.e. if wp is set to 300n, both PMOSs have widths of 300n – the procedure does not account for two FETs in series on a path.

3.3.6 Tri-State Inverter

The tri-state inverter consists of two PMOS and two NMOS FETs, all in series. The outside FETs (those closes to VDD/VSS) make up the actual inverter portion and the two middle FETs make up the enable portion. The tri-state inverter has the following parameters: wpINV, wpEN. wnEN. wnINV. lpINV. lpEN, lnEN, and lnINV. The "INV" designation refer the to the inverter FETs and the "EN" designation refers to the enable FETs. The schematic for the tri-state inverter can be found in Appendix A.

3.3.7 Bitcell

The bitcell used by ViPro is the typical 6 transistor bitcell with an inverter loop and two access transistors. The bitcell has an input pin for the word line and two input/output pins for the bit lines. A user can alter the widths and/or lengths of all FETs within the bitcell and the FETs are grouped by pull-up, pull-down, and pass gate devices. Thus, the bitcell has the following parameters: wpg, lpg, wpu, lpu, wpd, and lpd. The schematic for the bitcell can be found in Appendix A.

3.3.8 D Flip Flop

The D Flip flop used by ViPro consists of feedback tri-state inverters, a transmission gate, and inverters to generate the output. There is also a buffer to generate the clock signals needed for the tri-state inverters. ViPro has sized all FETs in the DFF specifically. Thus, there are no parameters to be altered in the DFF. The schematic for the D Flip Flop can be found in Appendix A.

3.3.9 Buffer Chain

The buffer chain is a series of inverters. The user specifies the number of inverters and the fan out of the subsequent stages. The initial buffer is minimum sized and each succeeding buffer is sized for the fan out specified by the user. The schematic for the buffer chain can be found in Appendix A.

3.3.10 Column MUX

The column MUX determines what data can pass onto/off of the bit lines. It contains two transmission gates (one for BL, one for BLB), two precharge FETs and one equalize FET for faster precharging of the bit lines. The bit width of the transmission gate and consequently the number of select lines depends on how many words are in the columns of the SRAM. For example, if there is one word per row, there is only a single transmission gate for BL/BLB and one select bit. However, if there are four words in a single row, there will be four transmission gates, which correlate to a four-bit wide transmission gate and two select bits. There is an upper level procedure that determines the bit width of select lines and correctly generates the desired column MUX. In either case, the column MUX has the following parameters: wpc, lpc, wpTX, lpTX, wnTX, and lnTX. The "pc" designation refers to the precharge/equalize FETs while the TX designation refers to the transmission gate FETs. The schematic for the column MUX can be found in Appendix A.

3.3.11 Sense Amplifier

The sense amplifier allows for faster reading of data from the bit lines. ViPro uses the typical cross-coupled inverters for its sense amp, along with some footer devices. The sense amp also employs a NAND-based SR latch to determine when the correct value has been read. The parameters of the sense amp are divided into the following device types: enable devices, equalize devices, cross coupled devices, input devices, and all precharge devices. Thus, the sense amp has the following parameters: wen, len, weql, leql, wpsa, lpsa, wnsa, lnsa, wbl, lbl, wsapc, and lsapc. The schematic for the sense amp can be found in Appendix A.

3.3.12 Eight Word Line Driver

ViPro implements an AND-NAND combination scheme to access eight word lines at any given moment. The eight word line driver consists of eight AND gates, eight NAND gates, and eight inverters. By initially ANDing the outputs of the two most significant predecoders, then NANDing the result with all eight

outputs of the least significant predecoder, and finally inverting these signals, the set of eight word lines are created. ViPro has designed all gates in the word line driver to be of minimum size. Therefore, there are no parameters to be altered. The schematic for the word line driver can be found in Appendix A.

3.3.13 Input/Output Block

The Input/Output Block contains the DFFs to hold the data in and data out bits and the tri-state inverters to send data to the bit lines. The I/O Block also has inverters to generate the enable signals for the tri-state inverters. The schematic of the I/O block can be found in Appendix A.

3.4 Intermediate Components (Leaf Nodes)

This section discusses the intermediate circuit components (leaf nodes) created from the lower level components, to continually build the hierarchy structure. The following sections clearly identify and discuss the individual leafs.

3.3.1 Bitcell Array

The bitcell array leaf node is an R rows by C columns array of bitcells, used to store the actual data. This procedure requires the bitcell with desired sizing to have already been made; it does not generate the bitcell. Thus, there are no sizing parameters to be altered at this level. This procedure merely places the bitcells in R rows and C columns.

3.3.2 Word Line Driver

The word line driver leaf node combines the outputs of the three 3:8 predecoders, by placing as many eight word line drivers as needed to access every sing word line. Similar to the bitcell array generator, this procedure simply places as many eight word line drivers as needed in order to access every single word line in the array. Therefore, there are no sizing parameters to be altered at this level.

3.3.3 Bitslice

The bitslice leaf node contains three components: the column MUX, the sense amplifier, and the input/output block. The individual components are previously generated and this procedure connects them together. There are no sizing parameters to be altered at this level in the hierarchy; the sizing of the three components must be specified when they are originally created.

3.3.4 Timing & Predecode

The timing & predecode leaf node contains multiple components: the timing block, the row predecoders, the column decoder, DFFs to store the row and column addresses and the write signal, and some basic gates to generate control signals. There are no sizing parameters to be altered in this circuit because this procedure just connects the individual components. All components must already be properly sized when being connected. The schematic of the timing & predecode can be found in Appendix A.

3.3.5 Buffer Chains – N/K Chains

The final leaf nodes are specific buffer chains, which the decoder optimization procedure determines if they are needed. These buffer chains are generated similarly to the buffer chain mentioned in Section 3.3.9. However, these buffer chains are used in a specific manner. The results of the decoder optimization procedure determine how many buffers, if any, are needed between the predecoder outputs and word line driver and between the word line driver and bitcell array; these buffer chains have been designated as the “N” and “K” chains, respectively. Thus, the user must specify the number of buffers and the fan out for each subsequent stage within the chain. The schematics for the N and K chains are similar to that of the buffer chain discussed previously. For further reference, see Section 3.3.9.

3.5 Complete SRAM Schematic

Once all leaf nodes have been created, a top level procedure connects all nodes together. This process easily executes by taking advantage of the hierarchy that has already been established throughout the entire schematic generation process. Since the top-level procedure is only connecting the leaf nodes together, there are no parameters that can or should be altered at this level. This procedure only requires input arguments of the number of rows and columns in the SRAM, the word size, the number of buffers in the N chain and the number of buffers in the K chain. The top level SRAM schematics can be found in Appendix A.

3.6 SRAM Block Diagram

Figure 6 shows the block diagram of the hierarchy structure created by the SKILL scripts. As previously mentioned, there three levels of hierarchy (top, intermediate, and lower levels). The top level contains the complete SRAM schematic. The intermediate contains the leaf nodes, which can vary between 4 and 6 components, depending on whether or not N/K buffers are needed. The lowest level contains all the gates, components and blocks. The lines in the figure designate which gates, components, and/or blocks are placed within each leaf node. Accordingly, all six leaf nodes are connected to the top level, complete SRAM schematic.

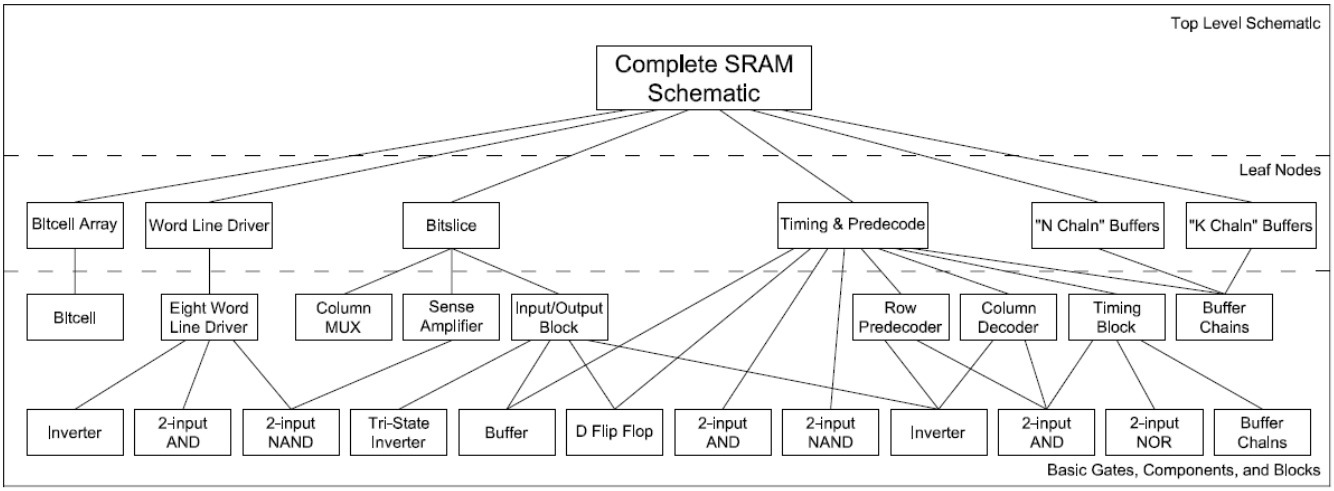

Figure 6: Block Diagram of Generated Hierarchy Structure.

3.7 Use of Scripts

To generate the complete schematic, the user must have the package of SKILL scripts that contain all the procedures needed for schematic generation. The package consists of 11 SKILL files (*.il) that must all be placed in a specific location. A complete list of these files can be found in Appendix B. The user must create a subfolder named "skill" within the folder where Cadence is run from, i.e. "/cadence/skill". All SKILL files must be placed within the "/skill" folder. However, the text file that contains all desired parameters and their values can be placed in any folder. Once all files have been placed within this folder, the user is ready to generate the schematics. Currently, users must independently generate the schematics, by running Cadence and loading the proper SKILL file. However, provisions will be added to ViPro so that it can automatically load the SKILL scripts and generate the schematics. Before running the top level script, the user must define/assign some variables. These include: (i) the text file to read the minimum FET sizes for the given technology, (ii) the text file to read parameters from, (iii) the library name to place these schematics, (iv) the letter case required for the given technology node, (v) the library name of the FETs to be used, (vi) the cell names of the PMOS and NMOS devices, and (vii) architectural specs, which include the number of rows, the number of address bits, the number of columns, and the length of a word. Now, the user can start Cadence, and in the Command Interpreter Window (CIW window), input the following command: load "./skill/darUvaEceSRAM_Schem.il" (making sure to include the quotations around the file path). An example of how to assign these variables can be found in Appendix C. For further reference and examples, see the Schematic Generation User Guide.

3.8 User Guide Documentation

There is an additional API-style documentation guide that contains every procedure used to generate the schematics. The documentation guide explains what each procedure does and what the arguments are and provides some examples of how to use the procedures.

3.9 Evaluation of Testing

Upon completion of creating the scripts, I found that creating a hierarchical structure is very important in organizing the components of the SRAM. The established hierarchy allows the user to descend through multiple levels of the SRAM to view certain portions and easily see the components and find problems. Also, creating CDF parameters for gates that will be re-used eliminated redundant schematic generation.

While automatic schematic generation can speed up the design process, it does limit design choices by slightly reducing user control. For example, since there are so many devices in the SRAM, it is difficult to completely address each transistor individually through an automatic schematic generator. So to simplify the problem, currently, all FETs must be of the same type, i.e. VTH, VTL, or VTG. Further work and future development of these scripts could potentially address each transistor individually so users could place specific types of FETs in certain places, but this would create lengthy argument inputs to the procedures. Another limitation of automatic schematic generation is the idea of being "stuck" with a certain design. While users have a knob to control FET sizes through a parameters file, the actual connections are hardcoded. Thus, users cannot simply move gates around and change some connections to test some methods. Some additional features to the script package could allow users to specify pre-made cells and have different types of a circuit. For example, a PDK may come with its own bitcell array. The scripts could be slightly modified so users can specify whether to create their own array or use the one that comes with the PDK. With respect to having different types of circuits, procedures could be added to the scripts to generate different schematics of the same circuit, i.e. different type of sense amps. This would give users greater flexibility during the design process.

Overall, I was pleased with the script package I have compiled. This script package generates the complete SRAM schematic assumed by ViPro and can adequately address the parameters of devices which ViPro deems necessary. While all gates, blocks and components are fully, individually parameterized, some future work can be done to fully parameterize the gates at the intermediate level in the hierarchy. For example, ViPro assumes the row and column decoders will be of minimum size (because there are the N/K buffers). However, by adding input arguments to the procedures that generate these decoders, a user could eventually alter the FET sizes of the AND gates and inverters within the decoders.

4. ACKNOWLEDGEMENTS

Domenic was responsible for the decoder optimization and Daniel was responsible for the automated schematic generation. We would like thank Professor Calhoun for his insight and advice during our time researching under his guidance. He has always inspired us to push for the best in all of our work. We would also like to thank Satyanand Nalam for allowing us to work on ViPro as our independent research project and always offering his assistance whenever we ran into problems. Without his support, we would not have achieved what we did.

5. REFERENCES

[1] http://www.cedcc.psu.edu/khanjan/vlsispec.htm. Retrieved on April 20, 2011.

[2] Schaller, R.R. (1997). Moore's Law: past, present, future. Spectrum, IEEE, 34(6), 52-59.

[3] Thompson, S.E. & Parthasarathy, S. (2006). Moore's Law: the future of Si microelectronics. Materials Today, 9(6), 20-25.

[4] Nalam, S., Bhargava, M., Mai, K., Calhoun, B.H. (2010). Virtual Prototyper (ViPro): an early design space exploration and optimization tool for SRAM designers. Proceedings of the 47th Design Automation Conference. 138-143.

[5] Carr, D. (2011). Decoder Optimization Plug-in for ViPro User Guide. Retrievable through UVA ECE Wiki. http://venividiwiki.ee.virginia.edu/twiki/bin/view/Main/ViPro

[6] Amrutur, B. S., & Horowitz, M. A. (2002). Fast low-power decoders for RAMs. Solid-State Circuits, IEEE Journal of, 36(10), 1506–1515.

[7] Markovic, D., Stojanovic, V., Nikolic, B., Horowitz, M.A., & Broderson, R.W. (2004). Methods for True Energy-Performance

Optimization. Solid-State Circuits, IEEE Journal of, 39(8), 1282-1293.

[8] Stojanovic, V., Markovic, D., Nikolic, B., Horowitz, M.A., & Broderson, R.W. (2002). Energy Delay Tradeoffs in Combinational Logic Using Gate Sizing and Supply Voltage Optimization. Solid-State Circuits Conference, Proceedings of the 28th European, 211-214.

[9] Reyno, D. (2011). Schematic Generation User Guide. Retrievable through UVA ECE Wiki. http://venividiwiki.ee.virginia.edu/twiki/bin/view/Main/ViPro